TAMPEREEN
AMMATTIKORKEAKOULU

KÄYTTÖLIITTYMÄTESTAUSAUTOMAATIO OSANA SOVELLUSKEHITYSPROSESSIA

Mikko Luhtasaari

Opinnäytetyö
Toukokuu 2018
Tietojenkäsittely
Ohjelmistotuotanto

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietojenkäsittely
Ohjelmistotuotanto

LUHTASAARI MIKKO:
Käyttöliittymätestausautomaatio osana sovelluskehitysprosessia

Opinnäytetyö 61 sivua, joista liitteitä 16 sivua
Toukokuu 2018

Opinnäytetyön tilaaja oli Cybercom Finland Oy. Heidän HSY-mittaridata -projektissaan oli noussut esille tarve automatisoida järjestelmätestausta. Opinnäytetyön tavoitteena oli tutkia, miten käyttöliittymätestausautomaatiota voitaisiin hyödyntää sovelluskehitysprosessissa. Tarkoituksena oli rakentaa käyttöliittymätestausautomaatio HSY-mittaridata-projektille.

Käyttöliittymätestausautomaatio voidaan luokitella kuuluvaksi järjestelmätestaukseen. Testaus olisi ollut hyvä ottaa osaksi prosessia jo varhaisemmassa vaiheessa projektia parhaiden tulosten saamiseksi. Selenium WebDriver osoittautui erinomaiseksi frameworkiksi laajemman käyttöliittymätestausautomaation rakentamisessa. Google Chrome -selain valittiin sopivimmaksi testiselaimeksi sen tarjoaman headless-tilan ja mobiiliemuloinnin vuoksi. CI-palvelimella on mahdollista suorittaa testejä ilman näytönohjainta.

Raportointia ja seurantaa tulisi parantaa resurssien käyttämisen tehostamiseksi. Toisaalta frameworkin valinta tulisi suorittaa aina nykytila huomioiden frameworkien nopean kehityksen vuoksi. Erilaisten testi-frameworkien tarjoamiin ominaisuuksiin tulisi tutusta paremmin. Nykyaikaisten JS-frameworkien toimintaa tulisi selvittää enemmän, ja sen pohjalta laatia yleispätevämpiä ratkaisuja epävakauden korjaamiseksi.

Asiasanat: käyttöliittymätestaus, testaus, selenium, jenkins, junit

ABSTRACT

Tampere University of Applied Sciences
Business Information Systems
Software Development

LUHTASAARI MIKKO:
User Interface Test Automation in Continuous Integration Development

Bachelor's thesis 61 pages, appendices 16 pages
May 2018

The bachelor's thesis was done for Cybercom Finland Oy. Their HSY-meter data project needed test automation to help with the software development process. The goal of this thesis was to research how to implement user interface test automation into a modern software development process. The purpose was to build user interface test automation for the meter data project.

User interface is categorized as system testing. It should have been implemented earlier in the project for the best results. Selenium WebDriver turned out to be an excellent framework for building the user interface test automation. Google Chrome was chosen as the best browser for testing because it supports both mobile emulation and headless-mode. It is possible to run user interface tests on a CI-server without graphics card.

Reporting and follow-up should be improved to allow for better planning and resource allocation. When choosing the framework for user interface testing, one should always do their own research. Frameworks evolve at such a rapid pace and sometimes it comes down to personal programming language preferences. Other test frameworks besides JUnit should be researched more. JavaScript based frontend frameworks require more research to enable more stable tests in the future.

Key words: user interface testing, testing, selenium, jenkins, junit

SISÄLLYS

LYHENTEET JA TERMIT

API-kirjasto	Rajapinta kutsujen tekemiseen
Change Control Board	Komitea, joka päättää sovelluksen muutoksista
CI-palvelin	Jatkuvan integraation palvelin
CSS	Verkkosivun tyylitiedosto
DOM	Verkkosivun komponenttien hierarkiapuu
Gradual Implementation	Vähittäisten muutosten vieminen tuotteeseen
Hotfix	Virheen korjaava pieni päivitys
HTML	Verkkosivun mallinnuskieli
IDE	Ohjelmointiympäristön tarjoava sovellus
Java	Ohjelmointikieli
JVM	Java-pohjaisten sovellusten suoritusympäristö
Label	Tarkentaa versionumeroa
Master-slave	CI-ympäristön malli, jossa isäntäkone käynnistää muita koneita
Maven	Java-projektien rakentamistyökalu
Parallel Implementation	Julkaisumalli, jossa uutta ja vanhaa versiota ajetaan rinnakkain
Phased Implementation	Julkaisumalli, jossa vähitellen päivitetään vanhaa
Rebase	Versionhallinnan toiminto, jolla tehdyt muutokset tuodaan vanhojen muutosten päälle
SPA	Sivusto, jossa muokataan DOMia koko HTML-dokumentin lataamisen sijasta
Scraping	Datan hakemista verkkosivuilta
Xpath	Tapa paikantaa elementtejä verkkosivulta

1 JOHDANTO

Cybercom on vuonna 1995 perustettu IT-konsulttiyritys. Cybercomin päätoimipisteet sijaitsevat Suomessa, Ruotsissa ja Tanskassa. Näiden lisäksi Puola ja Intia tuottavat tukitoimintoja Pohjoismaisille toimipisteille. Suomessa merkittävimpiä asiakkaita ovat muun muassa Kone, SOK, Alma Media sekä Väestörekisterikeskus.

Kansallinen palveluarkkitehtuuri lyhennetään työssä KaPa. KaPa-projektilla on pyritty yhtenäistämään ja helpottamaan julkishallinnon kokonaisarkkitehtuuria. KaPa-projektiin liittyen Cybercom on toteuttanut HSY:lle vesimittaridata-palvelun, jossa HSY:n asiakkaat voivat syöttää sähköisesti omia mittarilukemiaan. Nyt palveluun ollaan liitetty mukaan Valtuusrekisteristä tulevat valtuustiedot.

Palvelun kasvaessa on noussut esille tarve automatisoida testaus osaksi sovelluskehitysprosessia. Lähtötilanteessa testaus hoidetaan muutamalla yksikkötestillä, jotka ajetaan projektin rakentamisen yhteydessä. Yksikkötesteillä on kuitenkin melko vaikea testata tilannetta, jossa valtuustiedot tulevat Valtuusrekisteristä. Käyttöliittymätestauksella sen sijaan on helppo testata todellisia tilanteita.

Opinnäytetyön tavoitteena on tutkia, miten käyttöliittymätestausautomaatiota voidaan hyödyntää sovelluskehitysprosessissa. Työn tarkoituksena on automatisoida HSY:n mittaridatapalvelun käyttöliittymätestaus ja liittää se osaksi sovelluskehitysprosessia.

Työstä tulee hyötymään erityisesti kustannusyksikkö, jolle työ toteutetaan. Yksikössä tehtävä käyttöliittymätestaus on tehty tähän mennessä käsin. Työn tulisi helpottaa muiden projektien käyttöliittymätestauksen käyttöönotossa. Työssä syntyvä testausautomaatio vähentää projektin vaatimaa käsin suoritettavaa testausta.

2 Mitä testaus on?

2.1 Mistä virhetilanteet johtuvat?

Homes (2011, 3) kirjoittaa virheiden ja niistä syntyvien virhetilanteiden johtuvan joko ihmisen tekemisistä tai kohteen ja ympäristön vuorovaikutuksesta. Määrittelyvaiheessa tapahtuvalla kommunikaatiokatkoksella voi olla hyvin suuri vaikutus lopputuotteeseen. Toisaalta komponentti saattaa ylikuumeta liiallisen pölyn vuoksi tai satelliitti vaurioitua aurinkomyrskyssä. (Homes 2011, 3.)

Vaikka tyypillisesti vain lopputuote mielletään vialliseksi, voi myös määrittelydokumentti olla viallinen. Suuri osa virheistä saattaa olla lähtöisin juuri virheellisestä dokumentaatiosta. Virheiden ja viallisten tuotteiden välttämiseksi voidaan henkilöstöä kouluttaa sekä luoda organisaation laajuisia hyviä käytänteitä. Testaamalla pyritään löytämään näitä virheitä ennen kuin ne näyttäytyvät loppukäyttäjälle. (Homes 2011, 4.)

Homes (2011, 4) kertoo järjestelmien muuttuvan jatkuvasti monimutkaisemmiksi, ja toisaalta ihmisten tarvitsevan entistä enemmän järjestelmiä päivittäisessä elämässään. Esimerkiksi pankkiliikenteen vikatilanne saattaa vaikuttaa miljoonien ihmisten elämään radikaalisti. Koska turvaudumme sovelluksiin ja järjestelmiin entistä enemmän, on niiden toiminta pakko varmentaa testaamalla. Testauksen perimmäinen tarkoitus on varmistaa lopputuotteen toiminta. (Homes 2011, 4.-5.)

2.2 Testaus ja laadunvalvonta

Testausta usein luullaan virheellisesti laadunvalvonnaksi, vaikka se ei sitä ole. Testien tavoitteena on löytää virheitä ennen kuin tuote julkaistaan markkinoille. Laadunvalvonta voidaan mieltää pikemminkin prosessiksi. Sen tarkoituksena on löytää hyviä käytänteitä, ja niiden avulla parantaa prosessia tulevien virheiden välttämiseksi. (Homes 2011, 5.)

Homes (2011, 6) kertoo ihmisten kuuluvan kahteen leiriin testauksen suhteen. Toisen ääripään mielestä testauksen täytyy perustua puhtaasti dokumentaatioon ja sovellusta tulisi tarkastella hyvin analyyttisesti. Osan mielestä taas testauksen tulisi olla ketterää ja perustua riskiarviointeihin. (Homes 2011, 5.)

Homesin (2011, 6) mukaan analyyttinen lähestymistapa soveltuu suuriin projekteihin ja ketterä lähestymistapa pienempiin projekteihin. Jokainen projekti on kuitenkin yksilö, ja projektille tulisikin soveltaa sille parhaiten soveltuvia lähestymistapoja. Pienessäkin projektissa on mahdollista testata tuotetta dokumentaatiota vastaan, mikäli testit rakennetaan riittävän ketteriksi. (Homes 2011, 6.)

Virheet eivät synny itsestään, vaan niitä ilmestyy tuotetta aktiivisesti työstettäessä. Virheiden juurisyitä on monia erilaisia. Kommunikaatiovaikeudet asiakkaan kanssa saattavat aiheuttaa virheellisiä ominaisuuksia tuotteeseen. Huonot käytänteet ja ohjelmoijan kokemattomuus voivat johtaa virheelliseen lähdekoodiin. Kehitystiimin stressi ja kiire edesauttavat edellä mainittujen riskien kasvamisessa. (Homes 2011, 8.)

2.3 Testauksen paradoksit ja perusperiaatteet lyhyesti

2.3.1 Testaus voi todistaa vain virheiden olemassaolon

Testaamalla voidaan todistaa virheen olemassaolo, mutta ei niiden puuttumista. Virheen löytyminen on todiste sen olemassaolosta. Toisaalta virheiden löytymättömyys ei taas ole riittävä todiste siitä, että virheitä ei ole. (Homes 2011, 11.)

Jotta voitaisiin todistaa, että virheitä ei ole, tulisi testata kaikki mahdolliset sovelluksen sisällä tapahtuvat interaktiot. Testaaminen mahdollistaa virheiden mahdollisuuden minimoimisen, muttei niiden täydellistä poissulkemista. (Homes 2011, 11.)

2.3.2 Kaiken testaaminen on mahdotonta

Homes (2011, 11) nostaa esille yksinkertaisen laskimen testaamisen: pelkän laskimen kaikkien mahdollisten tapahtumien testaaminen on mahdotonta, joten suuremman sovelluksen testaus on myöskin mahdotonta. (Homes 2011, 11).

Testaamisen tulee olla järkevässä suhteessa siitä aiheutuviin kustannuksiin. Tästä syystä on pakko valita mitä testejä ylipäätään halutaan suorittaa. (Homes 2011, 11).

2.3.3 Varhainen testaaminen

Mitä aikaisemmassa vaiheessa virheet löydetään, sitä halvemmaksi niiden korjaaminen tulee. Mikäli virhe löydetään määrittelystä ennen ohjelmistokoodin kirjoittamista, vältetään ohjelmistokoodin korjaamisesta aiheutuvat kustannukset. Ohjelmistokoodin kustannuksiin sisältyvät lähdekoodin lisäksi myös testien suunnittelu ja toteutus. (Homes 2011, 12.)

Virheen korjaamiselle voidaan määritellä kustannus x, mikäli virhe löydetään jo määrittelyvaiheessa. Jos virhe huomataan ohjelmistokoodia kirjoitettaessa, on kustannus kymmenenkertainen. Mikäli virhe havaitaan vasta hyväksymistestauksessa, on siitä aiheutuva kustannus satakertainen. Tuotannosta löytyvä virheen hinta onkin jo tuhatkertainen. Kerrannaisvaikutusten vuoksi virhe voidaan korjata sitä nopeammin ja halvemmin mitä aikaisemmin se löydetään. (Homes 2011, 12.)

2.3.4 Testien toistuvuus

Testit esiintyvät tyypillisesti rykelminä. Mikäli jostain ominaisuudesta on löytynyt virhe, kannattaa jatkaa saman ominaisuuden testaamista pidemmälle. Virheet saattavat johtua esimerkiksi ohjelmoijan huonosta päivästä tai monimutkaisesta sovelluslogiikasta. (Homes 2011, 12.)

Homes (2011, 13) kertoo testien menettävän tehokkuuttaan ajan myötä. Jos samat testit ajetaan kerta toisensa jälkeen koko sovelluskehityksen ajan, alkavat ne menettää tehoaan, koska lähes kaikki niihin liittyvät virheet on jo löydetty ja korjattu. Testejä ja testidataa tulisikin päivittää samaa tahtia muun ohjelmiston kanssa. (Homes 2011, 13.)

Regressiotestaukseksi kutsutaan testausta, jossa vanhat testit ajetaan myöhemmin uudestaan. Tällä pyritään varmistamaan, etteivät uudet muutokset ole tuoneet takaisin vanhaa tunnistettua virhettä. (Homes 2011, 13.)

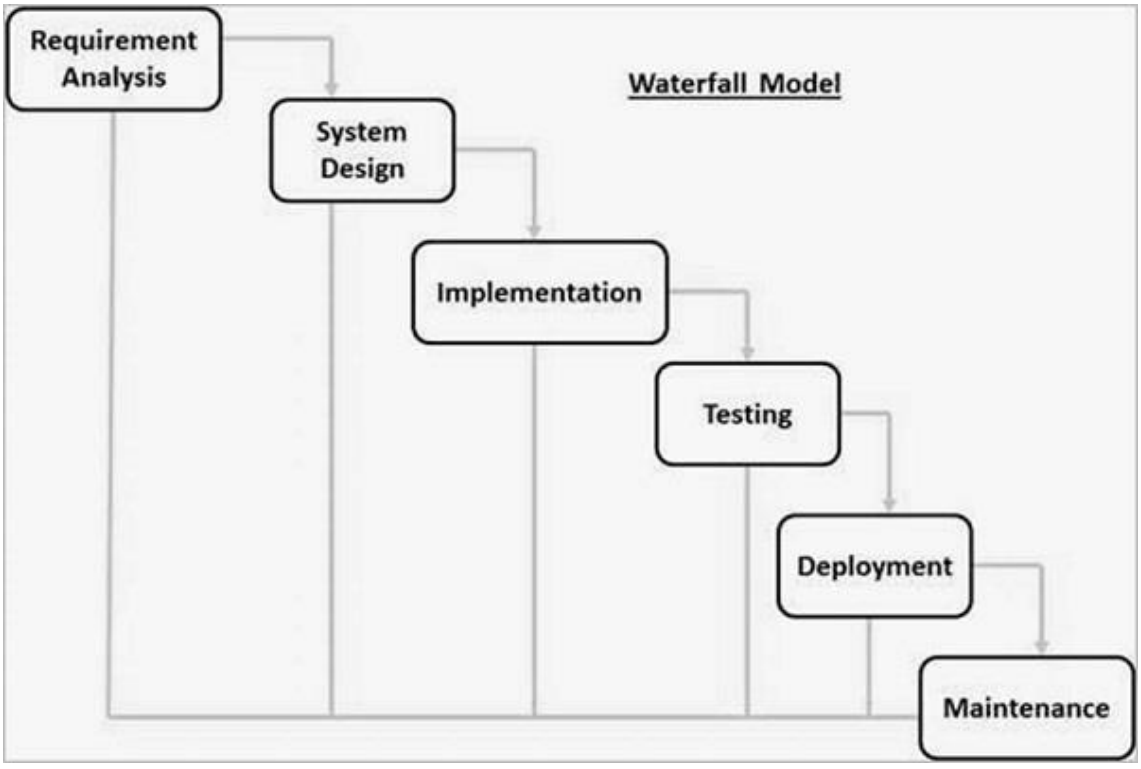
2.3.5 Testaus on kontekstiin sidottua

Homes (2011, 13) kirjoittaa testien suunnittelun perustuvan testien kirjoittajan näkemykseen sovelluksesta. Kehityksen alussa testaajalla saattaa olla erilainen näkemys sovelluksesta, joka kasvaa ja muuttuu kehityksen aikana. Muutokseen vaikuttavat vastuut ja löydetyt ongelmat. (Homes 2011, 13.)

Testien ja testaussuunnitelmien kierrättäminen toisista projekteista saattaa kuulostaa hyvältä idealta. Se ei kuitenkaan ole mahdollista, sillä jokainen projekti on yksilö, jolla on omat hyvät ja huonot puolensa. Tästä johtuen testit ja testaussuunnitelmat on aina kohdistettava tietylle projektille. (Homes 2011, 13.)

2.4 Testaus erilaisissa prosesseissa

2.4.1 Testaus vesiputousmallissa



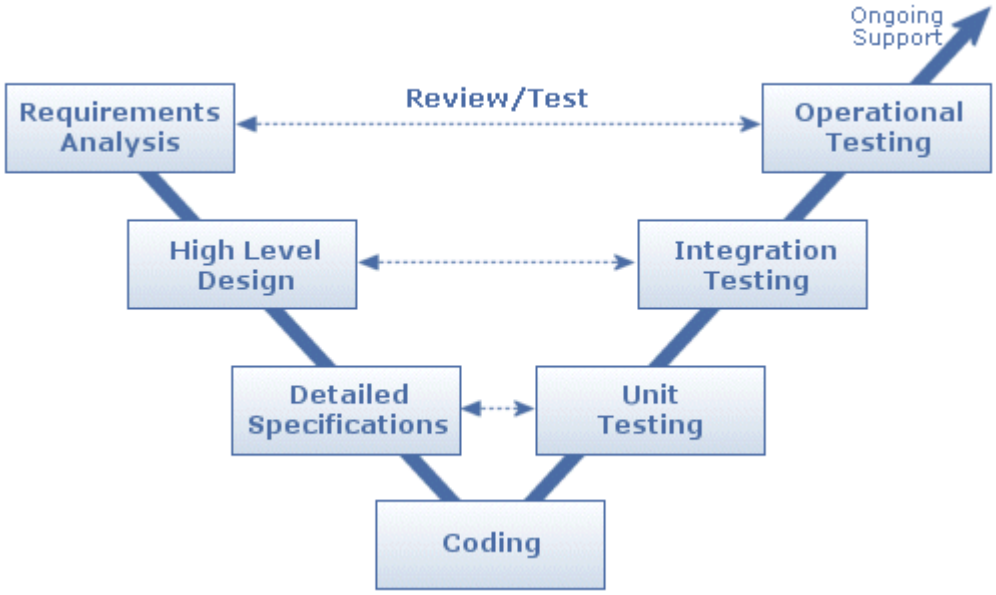
KUVA 1. Vesiputousmalli (SDLC - Waterfall Model)

Kuva 1. kuvastaa vesiputousmallia perinteisessä sovelluskehitysprosessissa. Vesiputousmalli oli ensimmäinen menetelmä hallita sovelluskehitysprosessia. Prosessin ajateltiin olevan lineaarisesti etenevä ketju erilaisia tapahtumia: kun edellinen vaihe on

valmis, voidaan siirtyä seuraavaan vaiheeseen. Prosessi etenee tarpeiden selvityksestä suunnittelun kautta toteutukseen. Toteutuksen jälkeen testataan ja julkaistaan tuote. Tämän jälkeen tuote siirtyy ylläpitoon. (SDLC - Waterfall Model.)

Vesiputousmallia on edelleen mahdollista soveltaa lyhyihin projekteihin, joissa vaatimukset eivät muutu. Myös käytettävän teknologian tulee olla tiimille tuttu, jotta tiedetään etukäteen teknologian asettamat vaatimukset. Malli on selkeä, ja sitä on helppo käyttää projektin johtamisessa. (SDLC - Waterfall Model.)

Vesiputousmallia kuitenkin kritisoidaan erityisesti sen joustamattomuudesta. Se ei kykene mukautumaan vaihtuviin vaatimuksiin, joten pitkään kestävät projektit eivät sovellu mallille. Ensimmäiset näkyvät osat tuotteesta saadaan hyvin myöhäisessä vaiheessa. (SDLC - Waterfall Model.)

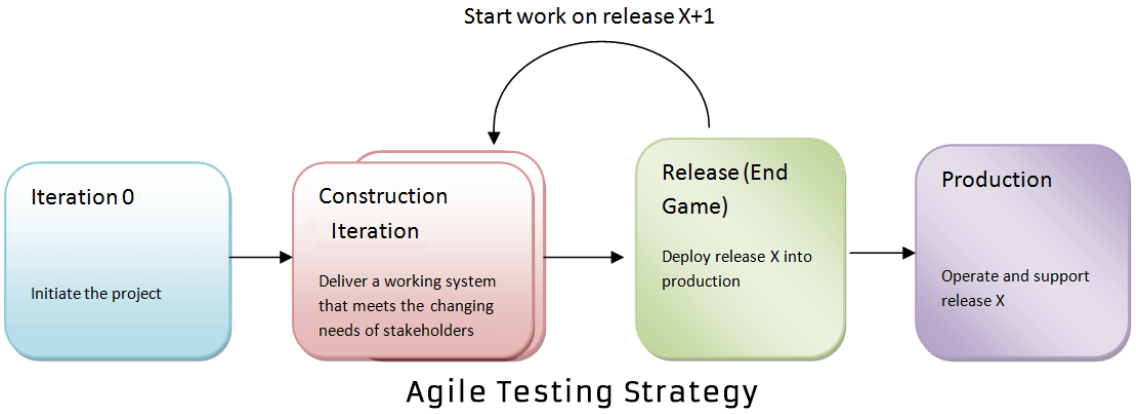


KUVA 2. Testaus vesiputousmallissa (The Death Of The V-Model)

Tyypillisesti vesiputousmallia kuvataan niin kutsutulla V-mallilla. Malli pyrkii kuvastamaan mitä testejä tulisi suunnitella minkäkin vaiheen jälkeen. Esimerkiksi heti tarkemman vaatimusmääritellyn valmistuttua tulisi suunnitella integraatiotestit. (The Death Of The V-Model.)

2.4.2 Testaus ketterissä menetelmissä

Ketterissä menetelmissä sovelluksen kehitys ja testaus tapahtuvat rinnakkain. Tarkoituksena on vähentää virheistä syntyviä kustannuksia löytämällä ne aikaisin ja toisaalta parantaa asiakastyytyväisyyttä luomalla laadukkaita tuotteita. *Agile Manifesto* julkaistiin vuonna 2001. Siinä nostettiin tiimi prosessin yläpuolelle ja korostettiin asiakkaan osallistamista. (Agile Testing – Overview.)



KUVA 3. Testaus ketterissä menetelmissä (Agile Testing Guide: Process, Strategies, Test Plan, Quadrants, Life Cycle)

Ketterissä menetelmissä sovelluksesta julkaistaan tiheästi uusia versioita, ja testaus mietitään uusiksi jokaisen julkaisun jälkeen. Koska kierto on hyvin lyhyt, pyritään testausta automatisoimaan niin paljon kuin mahdollista. Kuvasta 3. nähdään tyypillinen testauksen kierto ketterässä projektissa. Iteraatio 0:ssa kartoitetaan ja varataan testaamiseen liittyviä resursseja kuten henkilöitä ja työkaluja. *Construction Iteration* on vaihe, jossa suurin osa testauksesta tapahtuu. Testaus sisältää hyväksymistestaukselle tyypillisiä vähimmäisvaatimusten täyttymiseen liittyviä testejä, ja *Investigative testing* voi sisältää esimerkiksi tietoturvaan liittyvää testausta. Julkaisun kohdalla testaus keskittyy järjestelmä- ja hyväksymistestaukseen. Lopulta tuote päätyy tuotantoon. (Agile Testing Guide: Process, Strategies, Test Plan, Quadrants, Life Cycle.)

Ketterät menetelmät asettavat uudenlaisia haasteita testaajille. Testejä ei välttämättä ehditä suunnittelemaan kunnolla, koska syklit voivat olla vain viikon mittaisia. Koska vaatimusmäärittely elää koko prosessin ajan, ei välttämättä ole yksimielisyyttä toiminnallisuuden oikeellisuudesta. Testausautomaation rakentaminen on kallista ja sen ylläpidosta syntyy lisää kustannuksia. Testauksessa ei koskaan voida luottaa pelkkään automatiikkaan, vaan käsin testaamista on edelleen jatkettava tuotteen laadun varmistamiseksi. Myös testaajilta vaaditaan entistä parempia ihmissuhde- ja

kommunikaatiotaitoja. (Agile Testing Guide: Process, Strategies, Test Plan, Quadrants, Life Cycle.)

3 Testauksen päätasot

3.1 Hyväksymistestaus

3.1.1 Mitä on hyväksymistestaus

Hyväksymistestaus perustuu määrittelydokumenttiin. Sen tarkoitus on osoittaa, että käyttäjävaatimukset ovat toteutuneet. Hyväksymistestaus kannattaa suorittaa loppukäyttäjillä ja mahdollisimman todenmukaisessa ympäristössä. (Graig & Jaskiel 2002, 102.)

Loppukäyttäjien lisäksi hyväksymistestaukseen on syytä ottaa mukaan järjestelmätestauksen tekijöitä. Hyväksymistestaus alkaa, kun järjestelmätestaus on saatu päätökseen. Hyväksymistestauksen kielen ei tule olla liian teknistä, sillä kehittäjien lisäksi mukana on henkilöitä, joiden teknisen osaamisen taso ei ole välttämättä kovin korkea. (Graig & Jaskiel 2002, 103.)

3.1.2 Testauksen ajoitus

Hyväksymistestaus tulisi aloittaa heti korkean tason määrittelyn valmistuttua. Testauksen aloitus aikaisin on tärkeää, sillä hyväksymistestaus luo pohjan tuotteelle. Testitapaukset kertovat miten lopputuotteen tulisi käyttäytyä. (Graig & Jaskiel 2002, 103.)

Koska hyväksymistestauksen tarkoitus on selittää asiat yleisellä tasolla, ei sen tarvitse kestää kovinkaan pitkään. Hyväksymistestausta tehdessä testaus tehdään usein liian laajasti, ja päädytään samalla tekemään myös järjestelmätestausta. Hyväksymistestaus on mahdollista suorittaa jopa muutamassa päivässä. Nopeus perustuu siihen, että on melko nopeaa näyttää vaatimusten täyttyneen. (Graig & Jaskiel 2002, 103.-104.)

3.1.3 Käytettävyystestaus

Käytettävyystestauksella pyritään parantamaan loppukäyttäjän tyytyväisyyttä tuotteeseen ja tätä kautta tuotteen tuottavuutta. Tavoitteena on kerätä dataa, jonka perusteella voidaan tehdä tietoisia päätöksiä tuotteen suunnittelussa. Tällä pyritään luomaan tuote, joka on

hyödyllinen, tehokas, helppo oppia ja miellyttävä käyttää. (Rubin & Chisnell & Spool 2008, 21.-22.)

Käytettävyystestaus on kohdehenkilöiden tarkkailemista hallitussa tilanteessa. Siihen saattaa lisäksi liittyä haastatteluita sekä kvantitatiivisia mittauksia. Käytettävyystestaukselle on ominaista, että hypoteesien sijaan kehitelläänkin kysymyksiä, joihin halutaan vastauksia. Testauksen lopputuotoksena saadaan parannusehdotuksia tuotteen suunnitteluun. (Rubin & Chisnell & Spool 2008, 23.-25.)

Hyvin ja oikeaan aikaan suoritettuna käytettävyystestaus on erinomainen indikaattori ongelmista ja niiden mahdollisista ratkaisuista. Testaaminen vähentää riskiä julkaista epävakaa ja vaikeasti opittava tuote markkinoille. (Rubin & Chisnell & Spool 2008, 26.)

3.1.4 Julkaisutapoja

Pilotointi sekoitetaan usein betatestaukseen. Betatestaus tapahtuu aitoa ympäristöä mukailevassa ympäristössä. Pilotointi on tuotantokäytössä olevan järjestelmän testaamista pienellä kohderyhmällä. Virheet järjestelmässä saattavat vaikuttaa pilotoijien yrityksen toimintaan. (Graig & Jaskiel 2002, 116.-117.)

Gradual implementationissa tuote toimitetaan asteittain pienille käyttäjäryhmille. Tällöin kehittäjillä on aikaa reagoida ongelmatilanteisiin ja tehdä tarvittavat korjaukset ennen kuin tuotetta toimitetaan enempää. Tuotteen julkaisu kaikille käyttäjille kuitenkin hidastuu ja tuotteesta on ylläpidettävä useampaa versiota. (Graig & Jaskiel 2002, 117.)

Myös *Phased implementationissa* tuotetta toimitetaan asteittain käyttäjäryhmille. Erona *gradual implementationiin* on kuitenkin se, että jokaisessa julkaisussa toimitetaan uusia ominaisuuksia. Jo ensimmäiset käyttäjät saavat jollain tasolla käyttökelpoisen tuotteen ja saavat palautteellaan vaikuttaa tuotteen kehitykseen. Huonona puolena tarvittavan integraatiotestauksen määrä kasvaa huomattavasti. (Graig & Jaskiel 2002, 118.)

Parallel implementation tarkoittaa sitä, että uutta ja vanhaa tuotetta käytetään tuotantoympäristössä rinnakkain. Oletuksena uusi tuote on toimiva, kunnes toisin todistetaan. (Graig & Jaskiel 2002, 118.)

3.2 Järjestelmätestaus

3.2.1 Testaajat ja tiedonkeruu

Graig ja Jaskiel (2002, 121) kertovat järjestelmätestauksen tyypillisesti vievän eniten resursseja. Tuotteen ominaisuuksien testaamisen lisäksi siinä usein testataan kuormitusta, suorituskykyä ja luotettavuutta. (Graig & Jaskiel 2002, 121).

Mikäli projektilla on erillinen testausryhmä, on testausryhmän esimies useimmiten vastuussa testaussuunnitelman kirjoittamisesta. Hän on lisäksi vastuussa kehittäjien ja asiakkaiden palautteen käsittelemisestä. Jos erillistä testausryhmää ei ole, testaussuunnitelman kirjoittavat kehittäjät tai asiakkaat. (Graig & Jaskiel 2002, 121.-122.)

Järjestelmätestauksen testaussuunnitelman tulee toteuttaa laajemman testaussuunnitelman sille asettamat tavoitteet. Testaussuunnitelma rakennetaan sen mukaan, kuinka paljon muuta dokumentaatiota on käytettävissä. Tällaista dokumentaatiota voi olla esimerkiksi määrittelydokumentti. Testaussuunnitelmalle hyvä lähtökohta on hyväksymistestaussuunnitelma. Siinä määriteltyjen kohtien lisäksi tulee huomioida esimerkiksi asennukseen, tehokkuuteen ja lokalisointiin liittyvät asiat. (Graig & Jaskiel 2002, 122.)

3.2.2 Konfiguraationhallinta

Konfiguraationhallinta voidaan jakaa kahteen osaan. *Library management* kattaa ohjelmiston versioinnin. Ohjelmistolla varmistetaan, että tuotteesta julkaistaan ja kehitetään oikeaa versiota ja siihen liittyvää dokumentaatiota. (Graig & Jaskiel 2002, 123.)

Graig ja Jaskiel (2002, 123) kirjoittavat *Change Management Boardin* olevan lähempänä hallinnollista puolta. Sen vastuulla on määritellä, millä prioriteetilla ja aikataululla ohjelmistossa olevat virheet korjataan. (Graig & Jaskiel 2002, 123). Ketterässä mallissa virheiden priorisointi ja korjauksen aikataulutus on yleensä kehittäjien päätettävissä.

3.2.3 Aloitus

Järjestelmätestauksen aloituksen kriteerit tulee määritellä kustannusten säästämiseksi. Virheiden löytäminen järjestelmätestauksessa on kalliimpaa kuin niiden löytäminen integraatio- tai yksikkötestauksessa. Mikäli kehittäjä löytää työstään virheen yksikkötestauksen avulla, on hänen mahdollista korjata se ilman erillistä dokumentointia. Järjestelmätestauksessa löytynyt virhe sen sijaan on dokumentoitava. Tämän jälkeen virheelle on määriteltävä prioriteetti ja palautettava virhe kehittäjälle korjattavaksi. Kun virhe on korjattu, on se vielä testattava uudestaan. (Graig & Jaskiel 2002, 127.)

On myös ongelmallista, mikäli testattava järjestelmä muuttuu jatkuvasti. Tällöin tuotteen testaaminen käsin saattaa olla kustannustehokkaampaa, kuin kirjoittaa testejä jatkuvasti uusiksi. Mikäli testaajilla on ongelmia saada riittävän vakaata versiota testattavakseen, on mahdollista määritellä, että tuotteen on läpäistävä savutesti ennen kuin se hyväksytään järjestelmätestaukseen. (Graig & Jaskiel 2002, 128.-129.)

3.2.4 Savutestaus

Savutestaus on kokoelma testejä, joilla määritellään ohjelmiston toimivan riittävän hyvin normaalissa käytössä. Sen tarkoituksena on toisaalta antaa testaajille käsitys sovelluksen valmiusasteesta, ja toisaalta antaa kehittäjille palautetta sovelluksen vakaudesta. Testaajien on syytä rakentaa testit yhteistyössä kehittäjien kanssa. Tällöin myös kehittäjät saadaan sitoutettua savutestaukseen. (Graig & Jaskiel 2002, 129.)

Savutestauksen testien tulee olla pikemminkin laajoja kuin pikkutarkkoja. Niiden tavoitteena ei ole löytää jokaista virhettä sovelluksesta, vaan todistaa sen vakaus normaalissa käytössä. Kehittäjillä tulisi olla pääsy testiympäristöön, jotta he voivat itse kokeilla testien toimivuutta myös siellä. Vaikka testit menisivät läpi kehittäjän omalla koneella, ei se vielä takaa niiden onnistumista kehitysympäristössä. (Graig & Jaskiel 2002, 129.-130.)

3.3 Integraatiotestaus

3.3.1 Mitä on integraatiotestaus

Integraatiotestauksen tavoitteena on varmistaa, että tuotteen eri osat toimivat odotetulla tavalla yhdessä. Alimmalla tasolla kehittäjät varmistavat, että eri yksiköt toimivat yhdessä. Korkeammalla tasolla kehittäjät tai testaajat varmistavat järjestelmän eri palasten toimivan yhdessä. (Graig & Jaskiel 2002, 130.)

Graig ja Jaskiel (2002, 131) antavat esimerkin auton integraatiotestauksesta. Ilmaan ja polttoaineeseen liittyvät komponentit yhdessä muodostavat kaasuttimen. Kaasuttimesta ja muista komponenteista taas muodostuu moottori. Kaikkien näiden osien on toimittava yhdessä, jotta auto toimisi. (Graig & Jaskiel 2002, 131.)

3.3.2 Testaajat ja tiedonkeruu

Testaajien sijaan integraatiotestausta suorittavat kehittäjät. Kehittäjät myös varmistavat suunnitelmillaan, että ei synny päällekkäistä testausta. Integraatiotestauksen merkitys korostuu etenkin laajemmissa kokonaisuuksissa. (Graig & Jaskiel 2002, 132.).

Kuten kaikessa muussakin testauksessa tulisi integraatiotestaus aloittaa niin aikaisessa vaiheessa kuin vain mahdollista. Testaus on syytä aloittaa kaikkein kriittisimmistä ja riskialtteimmista komponenteista. Alhaisimmalla tasolla testataan eri yksiköiden välistä viestintää. Useimmissa yrityksissä tämä nähdään yksikkötestauksen jatkona. Myöhemmin myös laajemmat kokonaisuudet on testattava. Tämä vaatii useamman kehittäjän yhteistyötä. (Graig & Jaskiel 2002, 133.).

Korkean tason integraatiotestaus voi pohjautua arkkitehtuurisuunnitelmaan. On kuitenkin myös syytä huomioida määrittelydokumentin käyttötapaukset. Tiedonkeruussa on selvitettävä mitkä komponentit voidaan määritellä testiryhmäksi, mitkä ovat kriittiset toiminnot, kuinka paljon tulee testata ja mikä on integraatiotestauksen suhde järjestelmä- ja yksikkötestaukseen. (Graig & Jaskiel 2002, 134.).

3.4 Yksikkötestaus

3.4.1 Ongelmat ja koulutus

Yksikkötestaukseen kohdistuu usein asenneongelmia. Kehittäjät kokevat lähdekoodin tuottamisen olevan heidän tärkein työtehtävänsä. Optimistisesti jokainen kehittäjä kokee aina tekevänsä täydellistä lähdekoodia, jota ei tarvitse testata. Usein yrityksen palkitsemispolitiikkakin kannustaa ennemmin tuottamaan lähdekoodia kuin testaamaan sitä. (Graig & Jaskiel 2002, 137.-138.).

Kehittäjiä joudutaan usein kouluttamaan yksikkötestauksessa. Kehittäjien lisäksi heidän esimiehiään tulisi kouluttaa testaamisen perusteissa ja tekniikoissa. Projektipäälliköiden kouluttaminen osoittaa myös kehittäjille testaamisen tärkeyden. Kouluttamisessa kannattaa hyödyntää testauksesta vastaavan tiimin osaamista, mikäli sellainen löytyy yrityksestä. Testaaja voi kouluttaa henkilöstöä testien rakentamisessa ja testitapausten suunnittelussa. (Graig & Jaskiel 2002, 138.).

3.4.2 Standardointi

Helpoin tapa auttaa yrityksen testauksen standardien muotoutumisessa on tarjota esimerkit erilaisista dokumenteista. (Graig & Jaskiel 2002, 139). Graig ja Jaskiel (2002, 139) luettelevat seuraavat dokumentit vähimmäisvaatimuksiksi:

- Yksikkötestaussuunnitelma
- Testin suunnittelu
- Testin toteuttaminen
- Virheraportti
- Testiyhteenveto

Yleensä näiden lisäksi riittää muutama vuokaavio esimerkiksi virheen raportoinnista ja testien lokittamisesta. (Graig & Jaskiel 2002, 139).

Yrityksestä olisi hyvä löytyä henkilö, jonka näkemykseen testauksesta muut luottavat. Tämän henkilön ei tulisi olla johtotehtävissä vaan toimia muissa tehtävissä. Muiden työntekijöiden on hyvä tietää, että tämän henkilön puoleen voi kääntyä testaukseen

liittyvissä asioissa. Hänen tulisi myös olla mukana muuttamassa standardeja, jos olemassa olevat standardit todetaan toimimattomiksi. (Graig & Jaskiel 2002, 139.-140.)

3.4.3 Mittaaminen

Yksikkötestaus mahdollistaa yksikkökohtaisen mittaamisen. Tämä taas auttaa tunnistamaan ongelmallisia kohtia kehityksen alkuvaiheessa. Mikäli jostain paikasta on löydetty paljon virheitä, tullaan sieltä myös jatkossa löytämään paljon virheitä. Kyseessä saattaa olla monimutkainen järjestelmän osa, jota ei aivan täysin ymmärretä tai virheet saattavat johtua tehdyistä korjauksista. (Graig & Jaskiel 2002, 140.-141.).

Mittaaminen tuottaa yritykselle myös tulevaisuuden kannalta arvokasta tietoa. Sen pohjalta on mahdollista arvioida tulevien testauksien vaatimaa aikaa ja resursseja. Testausta tulisi kuitenkin mitata siten, etteivät kehittäjät koe heidän itsensä olevan mittauksen kohteena. (Graig & Jaskiel 2002, 141.)

4 Testausvälineen valinta

4.1 Robot Framework

Robot Framework on avoimen lähdekoodin testiautomaatio framework. Sitä käytetään hyväksymistestaukseen kehityksen ohella. Frameworkina se on moneen eri käyttötarpeeseen sopiva. (Bisht 2013, 26.)

Suurimpana erona muihin testauksen työkaluihin on Robot Frameworkin käyttämä syntaksi. Avainsana-pohjainen -syntaksi madaltaa vähemmän ohjelmoineiden ihmisten aloituskynnystä. Erilainen lähestymistapa testaukseen mahdollistaa testien tekemisen perinteisenä käyttöliittymätestauksena tai kuva-pohjaisena. (Bisht 2013, 26.)

4.2 NightmareJS

NightmareJS on JavaScript-pohjainen framework käyttöliittymätestaukseen. Se ei vaadi oikeaa selainta tai erillistä serveriä toimiakseen. Nämä ominaisuudet keventävät CI-palvelimen konfiguraatiota ja nopeuttavat testien ajamista. NightmareJS on kuitenkin mahdollista kehitysvaiheessa ajaa normaalissa selaimessa. (Blazing Fast Tests with Nightmare).

Huonoksi puoleksi voidaan laskea NightmareJS:n oma API-kirjasto. Mikäli kesken projektin havaitaan tarve kattavammalle kirjastolle, joudutaan testit kirjoittamaan suurimmilta osin uusiksi. NightmareJS mahdollistaa elementtien paikallistamisen sivustolla vain *CSS*-lokaattorien avulla. Tästä seuraa ongelmia, mikäli projektissa tulisi hakea elementtejä XPathin kautta. (Blazing Fast Tests with Nightmare).

4.3 Selenium WebDriver

Selenium WebDriver on paranneltu versio Selenium RC:stä. Selenium WebDriver tukee useita eri ohjelmointikieliä testien kirjoittamiseen. Tärkeimpinä ohjelmointikielinä voidaan mainita Java, Ruby, Python, Perl, PHP ja .NET. (Avasarala 2014, 13.)

Avasarala (2014, 13) luettelee Selenium WebDriverin tukemiksi selaimiksi muun muassa Mozilla Firefoxin, Internet Explorerin, Operan sekä Google Chromen. Mahdollisuus

testata useilla eri selaimilla on erityisen tärkeää, mikäli projektille on erikseen määritelty vähimmäisvaatimukset selaimille. Yhdellä testiympäristöllä usean selaimen testaaminen helpottaa testaustiimin työtä. (Avasarala 2014, 13.)

Seleniumiin on lisäksi olemassa oma IDE Firefoxin selain-laajennuksena. Laajennus voi helpottaa ensimmäisten testien tekemisessä. IDE nauhoittaa käyttäjän painallukset sivulla. Tämän jälkeen testiä voidaan muokata ja ajaa uudestaan. (Selenium HQ, Selenium IDE). Laajennus voi myös olla hyvä ratkaisu kehityksen alkuvaiheessa tapahtuvaan pienimuotoiseen testaamiseen.

4.4 Yhteenveto

Testivälineen valinnassa tulee huomioida testaajien tekninen tausta. Testaajien tausta mahdollistaa parhaiten Java- ja JavaScript-lähtöisen kehityksen, joten Robot Framework hylätään. Jäljelle jäävien NightmareJS:n ja Selenium WebDriverin välillä valinta voidaan tehdä niiden tarjoamien ominaisuuksien perusteella.

NightmareJS:n tärkeimmiksi ominaisuuksiksi nousivat nopeus ja ympäristön keveys. Nopeus ei ole projektissa ratkaiseva tekijä, sillä tulevat päivitykset tullaan ajamaan öisin. Päivittäminen taas laukaisee tarpeen testaukselle. Vähäiset ympäristövaatimukset CI-palvelimella ovat tavoiteltava tila, mutta ne eivät saa kuitenkaan nousta testauksen tarpeiden yläpuolelle.

Selenium WebDriverin tarjoama usean selaimen tuki on hyvä lisä projektille. Google Chromesta löytyvä mobiiliemulointi mahdollistaa pienimuotoisen mobiilitestauksen. IDE voi olla avuksi ensimmäisien testien luomisessa. Huonona puolena teknisen tuen tarpeessa täytyy turvautua Google-ryhmään.

Selenium WebDriverista löytyy kaikki testeissä tarvittava. Siitä on myös kirjoitettu useita kirjoja, joten tiedonhaku on helppoa. Selenium-projekti voidaan rakentaa Java-lähtöisesti pom.xml-tiedostosta. Tutkituista vaihtoehdoista Selenium WebDriver vaikuttaisi olevan paras valinta projektin käyttöön.

5 Selenium WebDriver

5.1 Projektin rakentaminen Ubuntu 16.04 käyttöjärjestelmässä

Projektin kehittäminen tehtiin Linux-ympäristössä, sillä se sopii hyvin Java-kehitykseen. IDE:nä toimi IntelliJ IDEA 2017.3.4 (Ultimate Edition). Kehitys tehdään Javalla, joten sen täytyy olla asennettuna tietokoneelle. Javasta käytössä oli versio 1.8.0_151.

SeleniumHQ (Selenium WebDriver) ohjeistaa Selenium WebDriverin rakentamisessa käytettäväksi Mavenia. Ensin luodaan tyhjä kansio projektille. Tähän luotuun kansioon sijoitetaan pom.xml -tiedosto. Maven kykenee lukemaan projektin riippuvuudet pom.xml -tiedostosta ja rakentamaan projektin sen perusteella. Liitteestä 1. löytyvä pom.xml asentaa kaikki alussa tarvitsemamme riippuvuudet projektille.

Komentoriviltä projektin voi rakentaa komennolla `mvn clean install`. Käytetyssä IDE:ssä IntelliJ IDEA:ssa oli mahdollista luoda Maven projekti pom.xml -tiedoston perusteella. Vastaavanlaisen ominaisuuden tulisi löytyä kaikista moderneista kehitysympäristöistä.

5.2 WebElements

5.2.1 Mitä ovat WebElementit

Verkkosivut rakentuvat erilaisista HTML-elementeistä. Tällaisia elementtejä ovat esimerkiksi linkit, painikkeet, lomakkeet, otsikot ja leipätekstit. Selenium WebDriverissa näitä elementtejä kutsutaan WebElementeiksi. (Avasarala 2014, 20.)

Elementti määrittyy alku- ja sulkutagista. Näiden väliin kirjoitetaan elementin sisältö. Esimerkiksi leipäteksti sisällöllä ”testi” näyttää seuraavalta. `<p>testi</p>` . Avasarala (2014, 21) kertoo käyttöliittymätestausautomaation olevan hyvin pitkälti näiden elementtien etsimistä ja erilaisten toimintojen suorittamista niille.

5.2.2 WebElementtien etsiminen sivustolta

SeleniumHQ (Selenium WebDriver) antaa esimerkin yksinkertaisesta elementin paikannuksesta. Oletetaan sivulta löytyvän elementti

```
<div id=coolestWidgetEvah>...</div> .
```

Kaikista tehokkain ja yksinkertaisin tapa paikantaa kyseinen elementti olisi seuraavanlainen.

```
WebElement element = driver.findEle-
ment(By.id(”coolestWidgetEvah”));.
```

findElement-metodi palauttaa viittauksen yksittäiseen WebElementtiin. *findElements*-metodi palauttaa Javan *List*-kokoelman kaikista elementeistä, jotka täyttävät hakuehdot. (Selenium HQ, Selenium WebDriver.)

ID:n lisäksi elementtien hakemiseen on useita muita erilaisia lokaattoreita. Avasarala (2014, 23) listaa seuraavat lokaattorit *Name*, *ID*, *TagName, Class, LinkText, PartialLinkText, XPath* ja *CSS* . SeleniumHQ (Location Strategies) suosittelee lokaattoreita käytettäväksi mahdollisuuksien mukaan seuraavassa järjestyksessä:

1. *ID*
2. *Name*
3. *XPath*
4. *LinkText, PartialLinkText*

Hyvin suunnitellun ja toteutetun sivun uniikit ID:t ja nimet ovat Selenium WebDriverille nopeimpia löytää. Lisäksi ne parantavat lähdekoodin luettavuutta. (Location Strategies). Elementtien sijainti voi vapaasti muuttua sivustolla, mikäli ne voidaan paikantaa ID:n tai uniikin nimen perusteella. Sen sijaan absoluuttisiin polkuihin nojaavat lokaattorit voivat mennä pienestäkin muutoksesta rikki.

5.2.3 WebElement actionien periytyminen

Koska WebElement on HTML-elementin ilmentymä, on niillä kaikilla samat käytettävät metodit. Ennen WebElementin metodikutsua suoritetaan vain tarkistus siitä, esiintyykö WebElement edelleen DOMissa. (Interface WebElement).

Avasarala (2014, 32) varoittaa, että vaikka kaikilta WebElementeiltä löytyvät samat metodit, eivät kaikki kuitenkaan pysty käyttämään samoja metodeja. WebElementtejä käsitellessä täytyykin pitää mielessä, mitä HTML-elementtiä se edustaa, ja valita metodikutsut sen mukaan. Esimerkiksi *<h1>*-elementtiä ei kannata klikata, vaan hakea sieltä tekstiä tai muuta tarpeellista tietoa.

5.2.4 Yleisiä WebElement actioneja

Avasarala (2014, 33) esittelee *sendKeys*-metodin, jolla voidaan simuloida käyttäjän näppäimistön painalluksia. Testejä tehdessä on kuitenkin paljastunut hyväksi käytännöksi tyhjentää käsiteltävä tekstialue ennen uuden syötteen lisäämistä. Tekstialueen tyhjennys onnistuu metodilla *clear*. (Avasarala 2014, 33.)

getText-metodi palauttaa WebElementin sisältämän, näkyvissä olevan tekstin. Mikäli tekstiä ei ole näkyvissä, ei metodi palauta mitään. (Avasarala, 2014. 39.) Testeissä varmistettiin oikean käyttäjän kirjautuminen seuraavanlaisella metodilla.

```
public boolean isUserCorrect(String name) {
        return userField.getText().contains(name);
}
```

Neljäs yleisesti käytetty metodi on *click*. WebElementin *click*-metodin käyttäminen on mahdollista, mikäli elementti on näkyvä ja sen korkeus ja leveys ovat suuremmat kuin 0 (Interface WebElement). Sivustolla liikkuminen on mahdollista suorittaa käyttäjän painalluksia simuloimalla.

5.3 Page Object Model (POM)

5.3.1 Miksi käyttää POMia

Testien lähdekoodia tulisi ylläpidettävyyden kannalta kohdella aivan kuten muutakin ohjelmistokoodia. Page Object Model -mallia noudattamalla testien ylläpidettävyys paranee. Itse testien ei tarvitse välittää sivun rakenteesta tai toimintaperiaatteista, sillä

sivujen ja testien lähdekoodi pidetään erillään. Koska toisto vähenee, tarvitsee sivun muuttuessa muutos tehdä vain yhteen paikkaan. (Gundecha 2012, 195.)

Testien tulisi käyttää Page Object Modeleita niin yleisellä tasolla, että niissä tapahtuvat muutokset eivät vaikuta testeihin mitenkään. Page Object Model tarjoaa rajapinnan, jonka avulla testit voivat käyttää sivua tietämättä sen tarkemmin sen toimintaperiaatteesta. (Gundecha 2012, 195.)

5.3.2 PageFactory ja @FindBy-annotaatio

Annotaatio on Java-ohjelmoinnille tyypillinen merkintä. Annotaation merkintä on esimerkiksi *@Test*. Java-kääntäjä injektoi annotaatioiden tilalle lähdekoodia kun ohjelma käännetään. Ohjelmoijalle annotaatiot ovat siis lyhyt ja ilmaisuvoimainen tapa tehdä monimutkaisiakin asioita.

Kappaleessa 5.2.2. esiteltiin erilaisia tapoja paikantaa WebElementtejä sivustolta. *@FindBy*-annotaatio hyödyntää samoja lokaattoreita, mutta eri kohdassa lähdekoodia. Aikaisemmin esitelty elementti

```
<div id=coolestWidgetEvah>...</div>
```

haettaisiin *@FindBy*-annotaatiolla

```
@FindBy(id = ”coolestWidgetEvah”)
private WebElement coolestWidgetEvah;
```

Ensimmäisellä rivillä ohjeistetaan WebDriveria etsimään WebElementtiä id:llä “coolestWidgetEvah”. Toisella rivillä se sijoitetaan privaattiin WebElement-muuttujaan nimeltä coolestWidgetEvah. (Avasarala 2014, 197.)

Toinen Page Object Modelille tärkeä luokka on *PageFactory*. Kun kaikki tarvittavat WebElementit on merkitty *@FindBy*-annotaatiolla, voidaan sivusta luoda ilmentymä PageFactoryn avulla. Staattisen initElements-metodin syntaksi on seuraava:

```
initElements(WebDriver driver, java.lang.Class
PageObjectClass)
```

(Avasarala 2014, 198.) initElements-metodia on hyvä kutsua luokan konstruktorissa. Liitteestä 2. löytyy esimerkki testeissä käytetystä Page Object Modelista luokalle *VETUPASLoginPage*.

5.3.3 POM palvelun tarjoajana

Avasarala (2014, 199) kirjoittaa verkkosivujen olevan pohjimmiltaan vain kokoelma erilaisia palveluita. Testeissä käytetty Valtuusrekisterin tarjoama toisen puolesta asiointi -palvelu on hyvä esimerkki tällaisesta ajattelusta.

Luokka *ValtuudetPickPersonOrCompanyPage.java* tarjoaa käytettäväksi kaksi metodia *findByNameButton* ja *isNamePresentInArray*. Näillä metodeilla testit voivat suorittaa tarkistuksia tai valita asioivansa jonkun tietyn henkilön tai yrityksen puolesta. Luokan tärkeimmät metodit löytyvät liitteistä 3. ja 4.

5.4 Headless-selaimet

5.4.1 Mitä tarkoittaa headless

Headless-tilassa testaaminen tarkoittaa selaimen ajamista ilman graafista käyttöliittymää. Tällöin ohjelmistokoodia ajetaan selaimen ohjelmistokoodia vasten ilman, että on kuitenkaan mahdollista nähdä mitä tapahtuu. (Colantonio 2017.)

Selaimia ajetaan tyypillisesti headless-tilassa verkkosivujen testaamiseksi. Muita yleisiä käyttötapauksia ovat JavaScript-kirjastojen testaaminen ja useamman käyttöliittymätestin yhtäaikainen ajaminen. (Arsenault 2018.) Colantonio (2017) täydentää listaa lisäämällä siihen vielä Scrapingin. Scraping tarkoittaa datan hakemista verkkosivulta. Esimerkkeinä tällaisesta datan hakemisesta hän antaa urheilutulosten hakemisen ja hintojen vertailun.

5.4.2 Headless-selainten tuomat hyödyt

HSY-mittaridata -projektissa tiedettiin jo alussa, että CI-palvelimella ei tule olemaan tukea graafiselle käyttöliittymälle. Aikaisemmin on ollut tarpeen luoda *master-slave* -järjestely, jossa *master*-kone käynnistää *slave*-koneen, joka sitten ajaa testit. Ratkaisu on kuitenkin hidas ja aiheuttaa lisäkustannuksia sekä lisää ympäristövaatimuksia. Ongelma on yleistynyt viime aikoina, joten siihen on olemassa parempikin ratkaisu.

Colantonio (2017) kertoo käyttäneensä headless-selaimia testien ajamiseen koneilla, joista puuttuu graafinen käyttöliittymä. Hän jatkaa kertomalla headless-selainten olevan 2-15 kertaa nopeampia kuin selaimet, joissa käytetään graafista käyttöliittymää.

5.4.3 Chrome headless-tilassa

Chrome-selaimeen saatiin headless-tuki versiossa Chrome 59, joka julkaistiin kesäkuussa 2017. Tämän myötä myös Seleniumia on mahdollista ajaa headless-tilassa Chrome-selaimella. Chromen tai Chromiumin lisäksi tarvitaan vain ChromeDriver, jotta Selenium testit on mahdollista ajaa headless-tilassa. (Bidelman 2017.)

HSY-mittaridata -projektissa päädyttiin käyttämään Chromea sen mahdollistaman headless-ajon vuoksi. Ympäristörajoitteena Chromen tulee olla asennettuna CI-palvelimelle. Koska CI-palvelimella oli Linux-käyttöjärjestelmä, käytettiin Chromen sijasta Chromiumia. Chromium on Chromeen perustuva open source -ohjelmisto. Chromen headless-tila vaatii selaimen lisäksi ChromeDriverin. Projektissa käytetty ChromeDriver-Linux-2.35. tallennettiin versionhallintaan muiden testiresurssien kanssa jaettuun kansioon. Ratkaisu helpotti testien ajamista lokaalisti useammalla koneella, mutta kasvatti versionhallinnan kokoa. Toinen vaihtoehto olisi ollut asentaa ChromeDriver CI-palvelimelle Chromiumin kanssa ja hakea kyseinen binääri ohjelmallisesti ympäristömuuttujiin.

5.5 Yleiset virhetilanteet

5.5.1 Single-page application

Single-page application eli SPA tarkoittaa verkkosivua tai -palvelua, jossa sivun sisältöä päivitetään lataamatta kuitenkaan koko sivua joka kerta uudestaan. Sivun renderöiminen tapahtuu perinteistä verkkosivua vahvemmin käyttäjän selaimella. Tarjolla on useita erilaisia JavaScript-kirjastoja ja -frameworkeja SPA-applikaatioiden toteuttamiseen. Näistä suosituimpia ovat Angular ja React. (Single-page Applications.)

HSY-mittaridata ei kuitenkaan ole aidosti SPA vaan lähempänä hybridi-sivustoa. Puhtaasti mittaridataan liittyvät toiminnot tapahtuvat yhdellä sivulla, mutta valtuuksia käytettäessä käyttäjä ohjataan Suomi.fi-valtuudet -palveluun. Tällä perusteella HSY-mittaridataa voidaan kutsua hybridi-sivustoksi.

Testejä rakennettaessa nousi nopeasti esille tarve varmistua lähtötilanteen oikeellisuudesta. Suomi.fi-palvelun tunnistautumisvälineet tarjoavat muutamia kaikille testiympäristössä toimiville kehittäjille yhteisiä testikäyttäjiä. Koska muutkin kehittäjät valtuuttavat ja poistavat valtuuksia käyttäjiltä, tulee ennen testien suorittamista varmistaa lähtötilanne Valtuusrekisteristä. Valtuusrekisterin verkkosivu on rakennettu AngularJS-frameworkilla. Valtuuksien käyttäminen taas tapahtuu asiointivaltuustarkastus-palvelun kautta, joka on React-pohjainen verkkosivu. HSY-mittaridata -palvelu taas on rakennettu ClojureScriptillä. Testeissä tuli siis huomioida kolme erilaista tapaa käsitellä sivun renderöimistä, mistä syystä testeissä törmättiin useisiin erilaisiin virhetilanteisiin.

5.5.2 Poikkeuskäsittely

APIn mukaan *WebDriverException* on *java.lang.RunTimeExceptionista* periytyvä ajonaikainen poikkeustilanne. Siitä periytyy useita muita WebDriverin poikkeuksia. (Class WebDriverException). Esimerkiksi *StaleElementReferenceException* kertoo siitä, ettei WebElementtiä ole enää olemassa DOMissa (Class StaleElementReferenceException).

Testien yhteydessä törmättiin useisiin erilaisiin poikkeustilanteisiin. WebDriver pyrkii suorittamaan testit mahdollisimman nopeasti, eikä aina ymmärrä mitä JavaScript-kirjasto on sivustolle tekemässä. Poikkeuskäsittely tehtiin Javan *try-catch* -logiikalla. Tyypillisesti *try*-osuus sisälsi toiminnon suorittamisen kymmenen kertaa odottaen toimintojen välissä 0,5 sekuntia ja lopettaen suorittamisen onnistumisen yhteydessä.

Catch-osuudessa otettiin kiinni poikkeustilanteet ja lokitettiin nämä poikkeukset. Liitteistä 5 ja 6 löytyvät esimerkit tällaisista metodeista.

6 Testien rakenne

6.1 JUnit 4

JUnit on tällä hetkellä suosituin testi-framework. Sen annotaatio-pohjainen lähestymistapa mahdollistaa elegantin lähestymistavan testaukseen. (Acharay 2014, 29.) JUnit on mahdollista liittää projektiin erillisenä .jar-tiedostona tai suoraan Maven riippuvuudella. Liitteestä 1. on nähtävissä JUnitin asentamiseen tarvittava riippuvuus pom.xml-tiedostossa.

Eräs frameworkin tarjoama ominaisuus on *org.junit.Assert*-paketti. Assertionit varmistavat, että niille syötetty tieto on oikeassa muodossa ja palauttavat poikkeustilanteen, mikäli ne havaitsevat eroja. (Acharay 2014, 29.). Liitteestä 7. voidaan nähdä esimerkki siitä kuinka metodia

```
Assert.assertFalse(failure message, condition)
```

voidaan hyödyntää testissä. *isNamePresent* palauttaa boolean-arvon, jonka tulisi onnistuneen testin tapauksessa olla false. Mikäli metodi palauttaa true-arvon, kaatuu testi ja sen voidaan päätellä epäonnistuvan. Virhetilanteessa *Assert.assertFalse* tulostaa sille ensimmäisenä parametrinä annetun virheviestin.

6.2 @Test-annotaatio

@Test-annotaatio luo mistä tahansa julkisesta metodista testin. Testit vertaavat lähes aina dataa odotusarvoihin, jotta voidaan varmistua ohjelman toimimisesta. Testeihin tyypillisesti liittyy siis datan alustamista, tietokantayhteyden luomista tai Selenium WebDriverin tapauksessa selaimen avaaminen. *@Before*-annotaatio suoritetaan ennen testiä, joten se on looginen paikka alustaa lähtötilanne. *@After*-annotoitu julkinen metodi taas suoritetaan testin jälkeen. (Acharay 2014, 31.)

HSY-mittaridata-projektissa testit periytetään abstraktista *AbstractTestBase*-luokasta. Luokkaan kuuluu kaikille testeille yhteisiä muuttujia esimerkiksi WebDriveriin ja tiedostojen sijaintiin liittyen. Se sisältää myös *@Before*-annotoidun *abstractSetup*-

metodin, joka löytyy liitteestä 8. Metodin tärkeimpinä tehtävinä on alustaa WebDriver halutuilla ominaisuuksilla sekä lukea testidata ulkoisesta tiedostosta.

Yksittäisestä testistä nähdään esimerkki liitteessä 9. Testin sisällä tulisi olla mahdollisimman vähän kovakoodattua sisältöä. Testien ylläpidettävyys paranee huomattavasti kun niiden toiminnallisuudet jaetaan erilaisiin metodeihin, joita myös muut testit voivat käyttää. *fillInAuth*-metodi on väliaikainen kehityksen metodi. Mikäli tarkastellaan mitä testissä tapahtuu, voidaan nopealla silmäyksellä sanoa, että kirjaudutaan sisään VETUPAS-tunnuksilla ja varmistetaan, että jotain tiettyä nimeä ei löydy. Koska käytettyjä metodeja on vain kaksi, on testin tarkoitus helppo ymmärtää.

Mikäli samassa luokassa on useampia testejä suoritetaan ne tyypillisesti lineaarisessa järjestyksessä. Suoritusjärjestys kuitenkin vaihtelee *JVM*-kohtaisesti. *JUnit* 4.11. versiossa eräs uusista annotaatioista on *@FixMethodOrder*, joka vastaa luokan sisäisten testien ajojärjestyksestä. `@FixMethodOrder(MethodSorters.NAME_ASCENDING)` ajaa testit aakkosjärjestyksestä a:sta alkaen. Tyypillisesti testien ei tulisi olla toisistaan riippuvaisia, mutta on kuitenkin mahdollista ajaa ne halutussa järjestyksessä. (Acharay 2014, 40.)

6.3 @TestSuite-annotaatio

Useamman testin peräkkäinen ajo on mahdollista *JUnit 4*:n tarjoamilla *Suite.class* ja *@Suite.SuiteClasses* -annotaatioilla. (Acharay 2014, 43). Voidaan luoda tyhjä *TestSuite*-luokka ja annotoida se *@RunWith(Suite.class)* ja `@Suite.SuiteClasses({luokka1.class, luokka2.class ..})`. *@Suite.SuiteClasses* ottaa parametreinä vastaan pilkulla eroteltuina suoritettavat luokat. *@BeforeClass*-annotoitu metodi suoritetaan ennen testin *@Before*-annotoitua metodia, ja vastaavasti *@AfterClass* suoritetaan, kun kyseisen luokan kaikki testit on suoritettu.

Liitteestä 10. on nähtävissä, miten HSY-mittaridata-projektissa käytettiin *@BeforeClass* ja *@AfterClass* -annotaatioita. *@BeforeClass* alustaa kaikille testeille yhteisen lokitustyökalun ja lukee testidatan ulkoisesta tiedostosta. *@AfterClass* vastaa lokitustyökalun kirjoittaman tiedoston viimeistelystä. Näin jokaisen *TestSuiten* sisältämän testin on mahdollista kirjoittaa lokitustaan samaan tiedostoon.

Testidataan voidaan lukea testikäyttäjien kirjautumis- ja osoitetiedot. Näiden lisäksi on testitapauskohtaisia tietoja kuten päivämääriä ja syötettäviä mittarilukemia. Mittaridata-projektissa päätettiin periyttää *TestSuite* abstraktista *AbstractTestSuiteBase*-luokasta. *AbstractTestSuiteBase*-luokka sisältää testihenkilöiden tietojen lukemisen properties-tiedostosta ja niihin liittyvät *Get*-metodit. Testitapaus kohtaisten tietojen lukemista varten *TestSuitella* on myös *Get*-metodi *properties*-tiedostolle. Periyttämisellä saatiin vähennettyä lähdekoodin toistoa ja selkeytettyä *TestSuitea* ylläpitoa helpottamaan.

7 Lokitus Javassa

7.1 Java lokituksen perusteet

Java julkaisi oman lokitustyökalunsa versiossa *JDK1.4.* (Gilstrap 2002). Javan lokitus perustuu kolmeen eri komponenttiin. *Logger* taltioi tapahtuman ja lähettää sen edelleen *Handlerille*. *Handler* muotoilee tapahtuman ulkoasun *Layoutin* avulla, ennen sen lähettämistä konsoliin, tiedostoon tai muuhun haluttuun paikkaan. Näiden lisäksi käytössä voi olla erilaisia *Filters*, jotka määrittelevät mitkä tapahtumat lähetetään eteenpäin. (Java Logging Basics.)

Javan tarjoaman *java.util.logging*-paketin lisäksi tarjolla on monia muitakin lokitus-frameworkeja, kuten esimerkiksi *Log4j*, *Logback* ja *tinylog*. Toisaalta projektille on mahdollista valita abstraktio-framework lokitukseen. Esimerkiksi *SLF4J* mahdollistaa lokituksen tekemisen ilman, että joudutaan määrittelemään lopullinen lokitus-framework. Tällöin lähdekoodin lokitus kirjoitetaan *SLF4J*:n avulla ja vasta ajonaikaisesti määritellään käytettävä framework. Mikäli frameworkia ei määritetä, osaa *SLF4J* ohittaa lokien kirjoittamisen. Lokitus-frameworkin valintaan vaikuttaa henkilökohtaiset mieltymykset ja projektin asettamat vaatimukset ja rajoitteet. (Java Logging Basics.)

7.2 Konfigurointi

Vaikka Javan lokitus-frameworkeja on mahdollista konfiguroida ohjelman suorituksen aikana, tehdään konfiguraatio tyypillisesti ennen ohjelman käynnistämistä ulkoisesta tiedostosta luettujen arvojen avulla. Javan tarjoama *java.util.logging*-paketti varastoi konfiguraationsa *logging.properties*-nimisessä tiedostossa. Projektikohtaista konfigurointia tehdessä, tulee käynnistyksen yhteydessä määritellä *java.util.logging.config.file*-tiedoston sijainti. (Java Logging Basics.)

Mittaridata-projektissa päädyttiin ratkaisuun, jossa käynnistyksen yhteydessä määritellään uusi *logging.properties*-tiedosto. Jokaiselle testille määriteltiin omalla rivillään lokitustaso. Mikäli testi epäonnistuu voidaan sen lokitustasoa laskea, jolloin saadaan enemmän lokitietoja mahdollisesta virheestä.

7.3 ExtentReports

Projektin alkuvaiheessa nousi esille tarve saada teknisen lokituksen lisäksi lokitusta asiakkaalle. Kehittäjien ja asiakkaan tarpeet raporttien suhteen eroavat suuresti, joten päädyttiin projektissa käyttämään kahta erillistä lokitus-frameworkia. Asiakkaan raportin tärkeimpänä ominaisuutena on ulkoasu. Hyväksi työkaluksi tyyliteltyjen ja helposti jaettavien raporttien tekemiseen osoittautui *ExtentReports*.

ExtentReportsista on olemassa ilmainen Community Version. Sen tärkeimpinä ominaisuuksina on HTML-pohjaisen raportin luominen ja kuvakaappausten liittäminen raportteihin. ExtentReports on mahdollista liittää projektiin Maven-riippuvuudella. (Pro vs Community Version.) Kuvakaappauksia ei kuitenkaan päätetty liittää asiakasraportteihin, sillä näytönohjaimen puutteen vuoksi sivustot näyttivät välillä tarpeettoman karuilta. Liitteestä 11. voidaan nähdä kuvakaappaus ExtentReportsin tuottamasta HTML-tiedostosta.

7.4 Kuvakaappaukset

Kuvakaappaukset tarjoavat arvokasta tietoa tilanteesta, jossa testi epäonnistuu. Ongelmaksi osoittautui se, että kuvakaappaus täytyi ottaa ennen kuin WebDriver ehti sulkeutua epäonnistuneen *Assertionin* tai jonkin odottamattoman *Exceptionin* seurauksena.

JUnitin TestWatcher-luokka sisältää erilaisia metodeja, jotka pitävät kirjaa testin etenemisestä. Sen metodeja kutsutaan esimerkiksi silloin, kun testi onnistuu tai epäonnistuu. (Class TestWatcher.)

Ylikirjoittamalla *TestWatcherin failed* ja *finished* -metodit pystyttiin säätelemään ohjelman toimintaa tilanteissa, joissa testi epäonnistuu tai päättyy. Liitteestä 12. nähdään, kuinka kuvakaappauksen tekeminen epäonnistuneen testin kohdalla toteutettiin. Toteutuksella saadaan päiväkohtaisiin hakemistoihin jokaisesta epäonnistuneesta testistä virheen hetkeltä kuvakaappaus.

8 Jatkuva integraatio

8.1 Mitä on jatkuva integraatio

Jatkuva integraatio on englanniksi *Continuous Integration* ja se lyhennetään kirjainyhdistelmällä CI. Kun työssä puhutaan jatkuvasta integraatiosta, tarkoitetaan nimenomaan CI-tyyppistä ajattelua.

Jatkuva integraatio on sovelluskehityskäytäntö, jossa kehittäjät yhdistävät työnsä mahdollisimman usein integraatio-haaraan. Tämän haaran pohjalta ohjelmistosta rakennetaan uusi versio. Tiheään tapahtuvalla yhdistämisellä pyritään välttämään yleisiä integraatioon liittyviä ongelma, jossa ohjelmiston eri komponentit eivät toimikaan yhdessä. (Pathania 2016. Continuous Integration.)

8.2 Hyviä käytänteitä

Kehittäjien tulisi työskennellä omassa yksityisessä haarassaan. Tällöin heillä on täysi vapaus kokeilla erilaisia lähestymistapoja ongelmiin ilman pelkoa niiden vaikutuksista koko järjestelmään. Mikäli käytetään versionhallinnan haaroja virheiden korjaamiseen, voidaan haarat nimetä virheen numeroinnin mukaan. Tällöin on nopeaa myös pitkän ajan jälkeen jäljittää missä ja miten virheet on korjattu. (Pathania 2016. Developers should work in their private workspace.)

Rebase tarkoittaa sitä, että muutosjoukko sijoitetaan toisen muutosjoukon päälle. Kehittäjien tulisi säännöllisesti ottaa integraatiohaaraan tehdyt muutokset itselleen ja ratkaista mahdolliset ristiriidat. Säännöllisesti tehtynä *rebase* vähentää riskiä tehdä muutoksia, joita ei ole mahdollista yhdistää muiden kehittäjien tekemiin muutoksiin. Toisaalta kehittäjien tulee myös viedä omat muutoksensa integraatiohaaraan. Muutosten vienti kerran päivässä vähentää muiden kehittäjien kohtaamia yhteensopivuusongelmia. On suositeltavaa, että jokainen integraatiohaaraan viety muutos käynnistää uuden version rakentamisen ja sen testaamisen. Tällöin on mahdollista saada jatkuvaa palautetta sovelluksen kehityksestä. (Pathania 2016. Rebase frequently from the mainline & Check-in frequently & Frequent build.)

Jokainen integraatiohaaraan viety muutos tulee testata. Tämän vuoksi on tarpeen rakentaa testausautomatiikkaa vähentämään testauksen vaatimia resursseja. Mitä lähempänä oikean elämän tilannetta testi on, sitä vaikeampaa sen automatisointi on. Automatisoitu testaus kuitenkin vähentää ihmisten tekemien virheiden määrää ja nopeuttaa tulosten saamista. (Pathania 2016. Automate the testing as much as possible.)

Rikkinäistä versiota ohjelmistosta ei tulisi koskaan viedä integraatiohaaraan. Kehittäjän tulisi aina ennen muutosten viemistä versionhallintaan tarkistaa version toimivuus omalla tietokoneellaan. Toinen vaihtoehto on ohjelmoida versionhallinta varmistamaan uuden version toimivuus. Versionhallinta voi tällöin hylätä muutokset, mikäli ne eivät tuota ehjää versiota. Usein jatkuvan integraation apuna käytetään myös staattista lähdekoodin analysointia esimerkiksi *SonarQubea*. (Pathania 2016. Don’t check-in when the build is broken.)

Perinteisessä julkaisumallissa erillinen julkaisutiimi huolehtii julkaisun tekemisestä. Julkaisutiimin täytyy tyypillisesti sopia testaustiimin kanssa sopivasta ajankohdasta testata julkaisua testiympäristössä. Koska julkaisu tehdään manuaalisesti, saattaa inhimillinen virhe viivästyttää kehittäjien palautteen saamista vuorokaudella. Julkaisu testiympäristöön tulisi automatisoida sen nopeuttamiseksi. (Pathania 2016. Automate the deployment.)

Julkaisujen numeroinnista tulisi olla selkeä ohjeistus, jota kaikki kehittäjät sitoutuvat noudattamaan. Versiolla tarkoitetaan järjestelmän kokonaistilaa juuri sillä hetkellä, ja *labelilla* viitataan pienempään kokonaisuuteen. Ajatellaan julkaisun numeroinnoin olevan muotoa xx.xx.xx: ensimmäiset kaksi numeroa viittaavat julkaisun numeroon, seuraavat kaksi virheiden korjauksiin ja viimeiset kaksi *hotfixien* numeroihin. (Pathania 2016. Have a labeling strategy for releases.) *Hotfix* tarkoittaa kriittisen virheen nopeaa korjaamista.

Jatkuvan integraation palvelun tulisi ilmoittaa muutoksista välittömästi. Esimerkiksi tekstiviestillä tai sähköpostilla saapuva ilmoitus julkaisun epäonnistumisesta mahdollistaa nopean reagoinnin siihen. Kaikista jatkuvan integraation työkaluista löytyy mahdollisuus välittömien ilmoitusten lähettämiseen. (Pathania 2016. Instant notifications.)

8.3 Testien ajaminen Jenkins-palvelimella HSY-mittaridata-projektissa

Projektin *Jenkins* vaati kaksi asennusta, jotta testit saatiin toimimaan. Nämä asennukset olivat *chromium-browser* ja *Xvfb*. *Chromium-browser* tarvitaan, jotta Selenium WebDriver kykenee avaamaan Chromium-selaimen. *Xvfb* eli *X virtual framebuffer* mahdollistaa graafisten tapahtumien simuloimisen virtuaalisessa muistissa, jota tarvitaan ruudunkaappausten tekemiseen. Kumpikin asennus kirjoitettiin *Dockerfileen* muiden riippuvuuksien joukkoon. Koska asennukset suoritettiin *apt-get*-komennolla, muuttavat ne myös ympäristömuuttujiin tarvittavat riippuvuudet.

Liitteestä 13. nähdään millaisella konfiguraatiolla versionhallinta yhdistettiin *Jenkins*-projektiin. Käytössä olleet hsy-jenkins -tunnukset generoitiin ja yhdistettiin projektin git-repositorioon *Read-only* -oikeuksin. Tarkoituksena oli, että mikäli SSH-avain vuotaa *Jenkinsistä*, ulkopuoliset eivät kuitenkaan pysty tekemään repositorioon muutoksia. *Branch Specifier* kertoo mistä haarasta lähdekoodi haetaan. Koska testejä kehitettiin erillisessä haarassa, on luonnollista myös hakea testit kyseisestä haarasta.

Liitteestä 14. voidaan nähdä projektin rakentamiseen ja ajamiseen käytetty shell-skripti.

```
”export DISPLAY=:99”
```

ohjaa *Xvfb*:n käyttämään ulostuloporttinaan porttia 99. Vaikka Jenkins palvelimella ei ole käytössä yhtäkään ulostulosporttia, on oletusportin vaihtaminen kuitenkin yleistettävämpi ratkaisu kuin oletusportin käyttäminen.

```
”cd testing ; mvn clean install -Djava.util.log-
ging.config.file=src/test/resources/logging.prop-
erties”
```

voidaan jakaa kolmeen osaan suorittamisensa suhteen.

1. *cd testing*
    a. Ohjaa hakemistoon testing.
2. *mvn clean install*

   a. Ohjaa Mavenin ensin tyhjentämään lokaalirepositorion ja tämän jälkeen asentamaan projektin.
3. *Djava.util.logging.config.file=src/test/resources/logging.properties*
   a. Ohjaa Mavenin ylikirjoittamaan *java.util.logging.config.file*:n tiedostolla, joka löytyy hakemistosta *src/test/resources/logging.properties*

Asiakasraporttien luomiseen käytetty ExtentReports lataa *CSS*-tyylitiedostoja muun muassa Googlen-palvelimilta. *Jenkinsin Content Security Policy Jenkins 1.641* -tietoturvapolitiikan vuoksi scriptien lataaminen ulkoisista lähteistä on kuitenkin kielletty. Ongelman ratkaisemiseksi Jenkinsiin on kuitenkin rakennettu *HTML Publisher plugin* -laajennus. Liitteestä 15. näkyy millaista konfiguraatiota HTML-raportin tallentamiseen käytettiin. Projektin rakentamisen ja siihen liittyvien testien ajamisen jälkeen laajennus tarkistaa määritellyn kansion ja luo sieltä löytyvien HTML-tiedostojen pohjalta raportin.

Liitteestä 16. nähdään miten testien ajaminen ja raportointi saadaan liitettyä muihin projektin osiin. Kun hsymitt-dev -projektista rakennetaan uusi vakaa versio, käynnistetään testien rakentaminen ja ajaminen tämän jälkeen. Testit on tarpeen ajaa vain silloin, kun versionhallintaan viedään uusi ja vakaa versio.

POHDINTA

Työn teoriaosuudessa käytiin lyhyesti läpi tärkeimmät testaukseen liittyvät termit. Näiden termien pohjalta voitaisiin sanoa käyttöliittymätestausautomaation olevan järjestelmätestausta. Testauksen tulisi olla mukana projektin alusta asti. Tutkimuksen laadullisesta luonteesta johtuen, tutkimustulokset eivät ole absoluuttisia totuuksia, vaan pikemminkin perusteltuja mielipiteitä. Toisaalta frameworkit muuttuvat ja kehittyvät jatkuvasti, joten testausvälineen valinta tulee tehdä aina nykytilanne huomioiden.

Testitulosten raportointi jäi hieman pinnalliseksi. Kahden erilaisen raportin lisäksi yrityksessä voitaisiin kerätä dataa testaukseen käytetystä ajasta suhteessa löydettyihin virheisiin ja mahdollisesti komponenttikohtaisista virheistä. Näin voitaisiin tulevaisuudessa paremmin suhteuttaa testaukseen käytettävät resurssit ja kiinnittää lisähuomiota ongelmallisiin komponentteihin.

Testi-frameworkin valintaan ei käytetty aikaa, vaan valittiin vain suosituin. Muut vaihtoehdot toisivat mukanaan erilaisia ominaisuuksia, ja voitaisiinkin harkita myös muiden frameworkien hyödyntämistä tulevissa projekteissa.

POM suhteessa *SPA*-sovelluksiin vaatisi lisätutkimusta. Selkeät ja hyvät käytänteet erilaisten JS-frameworkien kanssa parantaisivat tulevien projektien testien luotettavuutta. Esimerkiksi AngularJS:llä luotujen sovellusten lokaattorien käyttäminen vaatii lisätutkimusta testien ylläpidettävyyden parantamiseksi.

LÄHTEET

Acharya, S, 2014. Mastering Unit Testing Using Mockito and JUnit. Packt Publishing.

Arsenault, C, 2018. 6 Popular Headless Browsers for Web Testing. Luettu 11.4.2018. https://www.keycdn.com/blog/headless-browsers/

Avasarala, S, 2014. Selenium WebDriver Practical Guide. Packt Publishing.

Bidelman, E, 2017. Getting Started with Headless Chrome. Luettu 11.4.2018. https://developers.google.com/web/updates/2017/04/headless-chrome

Bisht, S, 2013. Robot Framework Test Automation. Packt Publishing.

Bunnybooboo, 2018. https://developer.mozilla.org/en-US/Firefox/Headless_mode

Code School. Single-page Applications. Luettu 11.4.2018. https://www.codeschool.com/beginners-guide-to-web-development/single-page-applications

CodeceptJS Team. Blazing Fast Tests with Nightmare. Luettu 6.4.2018. https://codecept.io/nightmare/

Colantonio, J, 2017. Headless Browser Testing Pros and Cons. Luettu 11.4.2018. https://www.joecolantonio.com/2017/09/21/headless-browser-testing-pros-cons/

Gilstrap, B, 2002. An Introduction to the Java Logging API. Luettu 24.4.2018. http://homepages.inf.ed.ac.uk/stg/teaching/ec/handouts/logging.pdf

Graig, R & Jaskiel S, 2002. Systematic Software Testing. Artech House.

Gundecha, U, 2012. Selenium Testing Tools Cookbook. Packt Publishing.

ExtentReports. Pro vs Community Version. Luettu 24.4.2018 http://extentreports.com/docs/versions/3/java/

Harmonic Software Systems, 2018. The Death Of The V-Model. Luettu 10.5.2018. https://harmonicss.co.uk/project/the-death-of-the-v-model/

Homes, B, 2011. Fundamentals of Software Testing. John Wiley & Sons, Incorporated.

JUnit. Class TestWatcher. Luettu 24.4.2018. https://junit.org/junit4/java-doc/4.12/org/junit/rules/TestWatcher.html

Kovalenko, D, 2014. Selenium Design Patterns and Best Practices. Packt Publishing.

Loggly. Java Logging Basics. Luettu 24.4.2018. https://www.loggly.com/ultimate-guide/java-logging-basics/

SeleniumHQ, 2018. Selenium WebDriver. Luettu 31.3.2018. https://www.seleniumhq.org/docs/03_webdriver.jsp

Pathania, N, 2016. Learning Continuous Integration with Jenkins. Packt Publishing.

Rubin, J & Chisnell, D & Spool, J, 2008. Howto Plan, Design and Conduct Effective Tests. John Wiley & Sons, Incorporated.

SeleniumHQ. Class WebDriverException. Luettu 11.4.2018. http://seleni-umhq.github.io/selenium/docs/api/java/org/openqa/selenium/WebDriverException.html

SeleniumHQ. Interface WebElement. Luettu 10.4.2018. https://seleniumhq.github.io/se-lenium/docs/api/java/org/openqa/selenium/WebElement.html

SeleniumHQ. Location Strategies. Luettu 6.4.2018 https://www.seleni-umhq.org/docs/06_test_design_considerations.jsp#location-strategies

SeleniumHQ. Selenium IDE. Luettu 6.4.2018. https://www.seleniumhq.org/projects/ide/

Tutorialspoint. Agile Testing – Overview. Luettu 10.5.2018. https://www.tutorialspoint.com/agile_testing/agile_testing_overview.htm

Guru99. Agile Testing Guide: Process, Strategies, Test Plan, Quadrants, Life Cycle. Luettu 10.5.2018. https://www.guru99.com/agile-testing-a-beginner-s-guide.html

Tutorialspoint. SDLC - Waterfall Model. Luettu 10.5.2018. https://www.tutorialspoint.com/sdlc/sdlc_waterfall_model.htm

LIITTEET

Liite 1. Projektin testien alkuperäinen pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                 xsi:schemaLocation="http://ma-
ven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>
        <groupId>fi.hsy.kapa-vesimittaridata</groupId>
        <artifactId>hsy.hsy-vesimittaridata-tests</artifac-
tId>
        <version>1.0</version>
        <properties>
            <selenium.version>3.9.1</selenium.version>
        </properties>
        <dependencies>
            <dependency>
                <groupId>junit</groupId>
                <artifactId>junit</artifactId>
                <version>4.12</version>
            </dependency>
            <dependency>
                <groupId>org.seleniumhq.selenium</groupId>
                <artifactId>selenium-server</artifactId>
                <version>${selenium.version}</version>
            </dependency>
        </dependencies>
</project>
```

Liite 2. VETUPASLoginPage.java

```
public class VETUPASLoginPage {
    private WebDriver driver;

    @FindBy(id = "hetu_input")
    private WebElement ssnField;

    @FindBy(id = "tunnistaudu")
    private WebElement tunnistauduButton;

    public VETUPASLoginPage(WebDriver driver) {
        this.driver = driver;
        PageFactory.initElements(driver, page: this);
    }

    public WebElement getSsnField() { return ssnField; }

    public WebElement getTunnistauduButton() { return tunnistauduButton; }
}
```

Liite 3. ValtuudetPickPersonOrCompanyPage.java getFindByNameButton

```
public WebElement getFindByNameButton(String name, Logger logger) {
    WebElement element = null;

    for (int i = 0; i < 10; i++) {

        try {

            if(i != 0) {
                Thread.sleep( l: 500L);
            }

            WebElement tbody = wait.until
                    (ExpectedConditions.visibilityOf
                            (driver.findElement(By.id("parties-table")).
                                    findElement(By.tagName("tbody"))));
            logger.log(Level.FINER, s: "Table is available");

            for (WebElement webElement : tbody.findElements
                    (By.tagName("tr"))) {

                for (WebElement webElement2 : webElement.
                        findElements(By.tagName("td"))) {

                    if (webElement2.getText().contains(name)) {
                        element = webElement;
                        i = 10;
                    }
                }
            }
        } catch (InterruptedException ie) {
            logger.log(Level.SEVERE, s: "Thread interrupted!", ie);
        } catch (WebDriverException wde) {
            logger.finer( s: "Element not yet ready");
        }
    }

    return element;
}
```

Liite 4. ValtuudetPickPersonOrCompanyPage.java isNamePresentInArray

```
public boolean isNamePresentInArray(String name, Logger logger) {
    boolean isPresent = false;

    for (int i = 0; i < 10; i++) {
        try {
            if (i != 0) {
                Thread.sleep( l: 500L);
            }
            WebElement tbody = wait.until(ExpectedConditions.
                    visibilityOf(driver.findElement
                            (By.id("parties-table")).
                            findElement(By.tagName("tbody"))));
            logger.log(Level.FINER,  s: "Table avaible");

            for (WebElement webElement : tbody.findElements
                    (By.tagName("tr"))) {

                for (WebElement webElement2 : webElement.
                        findElements(By.tagName("td"))) {

                    if (webElement2.getText().contains(name)) {
                        isPresent = true;
                        i = 10;
                    }
                }
            }
        } catch (InterruptedException e) {
            logger.log(Level.SEVERE,  s: "Thread sleep interrupted");
        } catch (WebDriverException wde) {
            logger.log(Level.FINER,  s: "Table not ready yet");
        }
    }

    return isPresent;
}
```

Liite 5. StaleElementReferenceException ja ElementNotVisibleException

```
public boolean clickByLocator(WebDriver driver, By locator) {
    int attempts = 0;
    boolean isOk = false;
    while (attempts < 50) {
        try{
            try{
                Thread.sleep( l: 100L);
            } catch (InterruptedException ie) {
                logger.log(Level.SEVERE,  s: "Thread interrupted!", ie);
            }
            driver.findElement(locator).click();
            isOk = true;
            break;
        } catch (StaleElementReferenceException sere) {
            logger.log(Level.FINE,  s: "StaleElement", sere);
        } catch (ElementNotVisibleException enve) {
            logger.log(Level.FINE,  s: "ElementNotVisible", enve);
        } catch (WebDriverException wde) {
            logger.log(Level.FINE,  s: "WebDriverException", wde);
        }
        attempts++;
    }

    return isOk;
}
```

Liite 6. WebDriverException

```
private boolean isAddressPresentInArray(String name, Logger logger) {
    boolean isNamePresent = false;
    WebElement table = null;
    for (int i = 0; i < 10; i++) {
        try {
            Thread.sleep( l: 500L);
            if(table == null) {
                table = driver.findElement(By.className("list-group"));
            }
            List<WebElement> buttons = new ArrayList<>();
            buttons.addAll(table.findElements(By.tagName("a")));
            numberOfAddresses = buttons.size();
            for (WebElement e : buttons) {
                for (WebElement td : e.findElements(By.tagName("td"))) {
                    if (td.getText().contains(name)) {
                        isNamePresent = true;
                        i = 10;
                    }
                }
            }

        } catch (InterruptedException | WebDriverException e) {
        }
    }
    return isNamePresent;
}
```

Liite 7. Assertoiminen JUnitilla

```
Assert.assertFalse( message: "Found a mandate for "
                + alandsbankenName +
                " when expected not to find one!",
        new MandateUtil().isNamePresent
                (driver, wait, alandsbankenName, logger));
```

Liite 8. Testin @Before-annotoitu metodi

```
@Before
public void abstractSetup() {
    testLogger = TestSuite1.getTestLogger();
    System.setProperty("webdriver.chrome.driver", USER_DIR);
    ChromeOptions chromeOptions = new ChromeOptions();
    if (System.getProperty(HEADLESS_FROM_PROPERTIES).equals("true")) {
        chromeOptions.setHeadless(true);
        chromeOptions.addArguments("window-size=1920x1080");
        chromeOptions.addArguments("--no-sandbox");
    }
    driver = new ChromeDriver(chromeOptions);
    driver.manage().window().maximize();
    wait = new WebDriverWait(driver, TIME_TO_WAIT);
    setup();
}
```

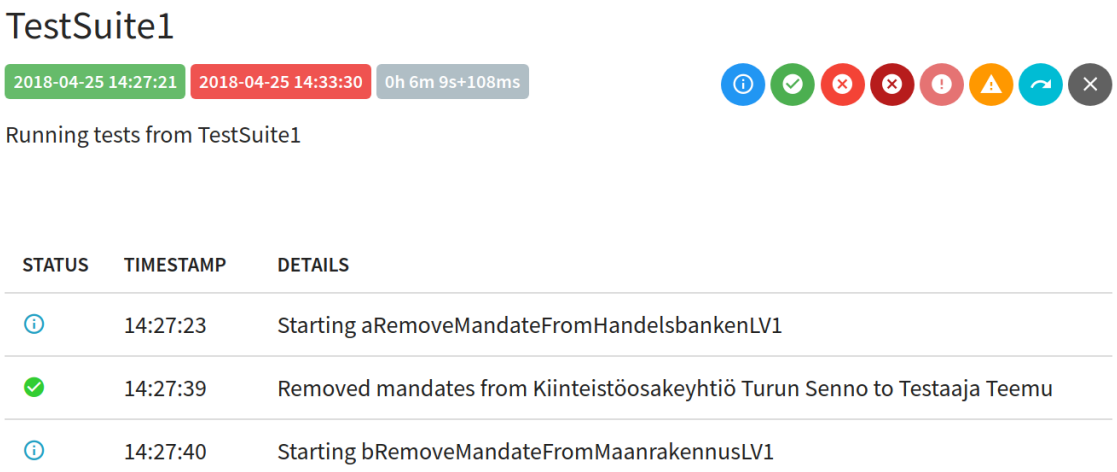
Liite 9. @Test-annotoitu-metodi

```
@Test
public void tryToUseWrongTypeOfMandateMD10() {
    logger.log(Level.INFO,  s: "Running test " +
            "tryToUseWrongTypeOfMandateMD10");
    testLogger.log(LogStatus.INFO,  details: "Starting " +
            "tryToUseWrongTypeOfMandateMD10");

    new LoginUtil().loginWithHandelsbanken(driver, wait,
             isHSYWatermeterPage: true, handelsbankenUsername,
            handelsbankenPassword, handelsbankenOTP, logger);
    logger.log(Level.FINE,  s: "Logged in as Handelsbanken");

    fillInAuth();

    Assert.assertFalse( message: "Found a mandate for " +
                    alandsbankenName + " when expected not to find one!",
            new MandateUtil().isNamePresent(driver,
                    wait, alandsbankenName, logger));

    testLogger.log(LogStatus.PASS,
             details: "Mandatees were processed correctly");
    logger.log(Level.INFO,  s: "Test passed");
}
```

Liite 10. @BeforeClass ja @AfterClass -annotoidut metodit

```
@BeforeClass
public static void setUp() {
    File f = new File(FILE_LOCATION);
    if(f.exists() && !f.isDirectory()) {
        report = new ExtentReports(FILE_LOCATION, replaceExisting: true);
    } else {
        report = new ExtentReports(FILE_LOCATION, replaceExisting: false);
    }
    readValuesFromProperties(prop);
    testLogger = report.startTest( testName: "TestSuite1",
            description: "Running tests from TestSuite1");
}
@AfterClass
public static void tearDown() {
    testLogger.log(LogStatus.INFO, details: "Test Suite closed");

    report.endTest(testLogger);

    report.flush();
}
```

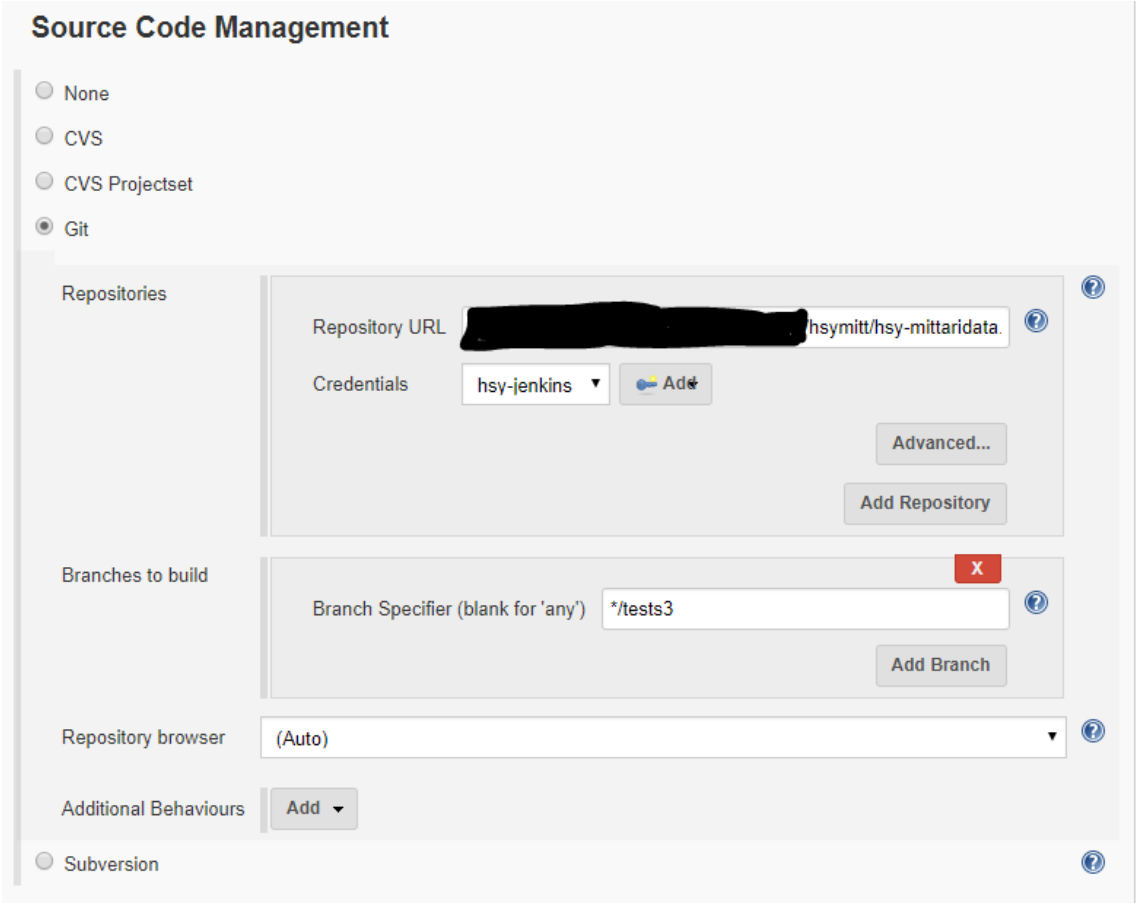
Liite 11. Kuvakaappaus ExtentReportsista



Liite 12. TestRulen ylikirjoittaminen

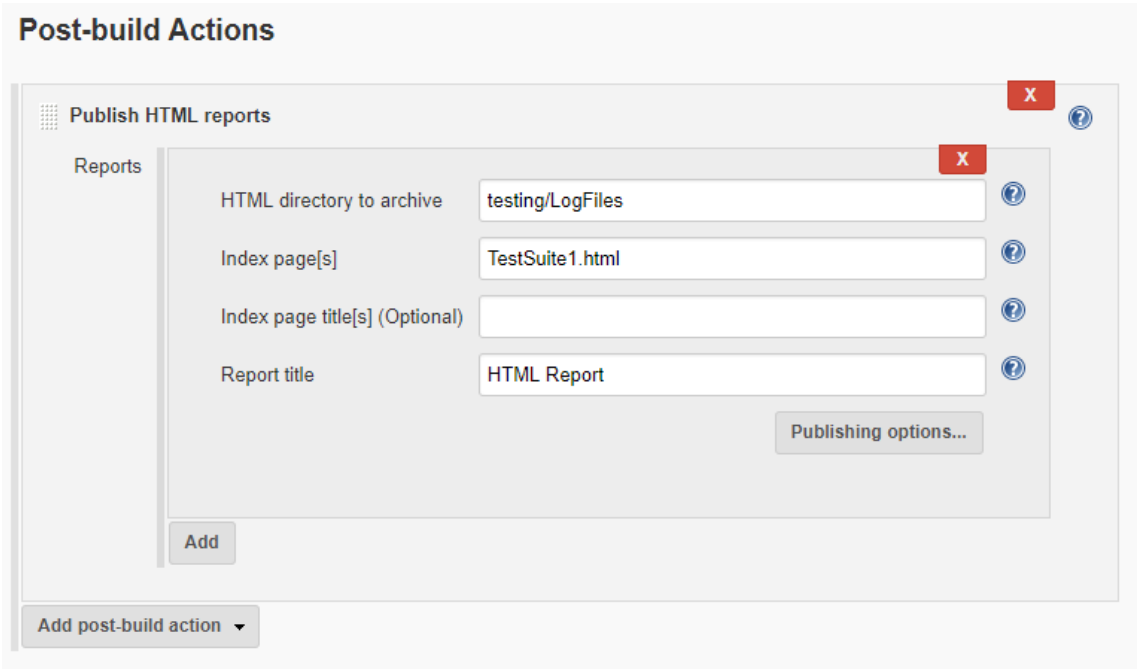
```
@Rule
public TestRule testWatcher = new TestWatcher() {
    @Override
    public void finished(Description desc) { driver.quit(); }

    @Override
    public void failed(Throwable e, Description d) {
        testLogger.log(LogStatus.ERROR,  details: "Test " + d.getMethodName()

        File file = new File(SS_LOCATION);
        if (!file.exists()) {
            if (file.mkdir()) {
                logger.fine( s: "Created a directory for screenshots");
            }
        }

        File scrFile = ((TakesScreenshot) driver).getScreenshotAs(
                OutputType.FILE);
        String scrFilename = d.getMethodName() + ".png";
```

Liite 13. Jenkinsin versionhallinnan konfiguraatio



Liite 14. Projektin rakentamisen Shell-skripti Jenkinsissä

Liite 15. HTML-raporttien julkaisu



Liite 16. Automaattinen testien ajaminen